An enterprise directory solution with DB2

by S. S. B. Shi
E. Stokes
D. Byrne
C. F. Corn
D. Bachmann
T. Jones

LDAP (Lightweight Directory Access Protocol) is a technology that can provide directory services to a range of applications. Directory service, a critical part of distributed computing, is the central point where network services, security services, and applications can form an integrated distributed computing environment. The simplicity of LDAP enables users to store and retrieve data easily from the directory. Nevertheless, as the use of directory services becomes more widespread, directories will need to scale to support millions of entries and millions of user requests with subsecond predictable performance. LDAP directories can be implemented using various storage mechanisms such as flat files, b-trees, or databases. This paper discusses an implementation of LDAP that uses the IBM DATABASE 2™ relational database as the data store and query engine to meet the directory service requirements. Performance analysis is provided to show that a relational database can be used to successfully meet the performance and scale needs of an LDAP directory while remaining secure and competitive with other vendor implementations.

Directories are special-purpose databases, usually containing categorized information to support frequent data retrieval and data update.[1] Directory service, a critical part of distributed computing, is the central point where network services, security services, and applications can form an integrated distributed computing environment. The current usage of a directory service can be classified into the following categories:

- Name service—Use directory as a source to locate Internet host address or the location of the server. Examples are Domain Name System (DNS) and DCE (Distributed Computing Environment) Cell Directory Service (CDS).
- User registry—Store information of all users in a system. A central repository of user information will enable the system administrator to administer the distributed system as a single system image, especially if the system is composed of a number of interconnected machines. Novell Directory Services** (NDS**) is an example.
- White pages lookup—Some modern e-mail clients provide users with the capability of looking up people's names and e-mail addresses. The users type in the name, or part of the name, and the directory service will extract the e-mail information for the user. Netscape Communicator**, Lotus Notes**, Eudora**, and other e-mail clients provide the address book lookup capability.

With more and more applications and system services demanding a central information repository, the next-generation directory service will be providing system administrators with a data repository that could significantly ease the administrative burden. In addition, the future directory service will also provide end users with a rich information data warehouse that allows them to access department or company employee data, or resource information such as the name and location of printers, copy machines, etc. In the Internet or intranet environment, users will be able to use the public key certificates stored in the directory to handle encrypted or digitally signed documents.

Lightweight Directory Access Protocol (LDAP) is an emerging technology that can provide directory services to applications ranging from e-mail systems to distributed system management tools. LDAP is a simple directory data access protocol that supports a rich set of operations for a wide range of applications. LDAP is an open Internet standard, defined by the Internet Engineering Task Force (IETF). A number of implementations of LDAP are available, ranging from commercial to publicly available open-source products.

One key feature of LDAP is it is simple but functionally rich. The simplicity of LDAP enables users to store and retrieve data easily from the directory. Nevertheless, as the use of directory services becomes more widespread, directories will need to scale to support millions of entries and millions of user requests with subsecond predictable performance. The protocol (LDAP) and the application programming interfaces (APIs) to access the directory are simple, but the amount of data stored in the directory can be tremendous. There must be a natural growth path for scale and performance that does not require an all-out replacement of the currently installed directory.

LDAP directories can be implemented using various storage mechanisms such as flat files, b-trees, or databases. The data store is as important as (if not more than) the protocol implementation of the directory server. It must scale, provide transactional integrity, be robust, and handle a variety of queries securely from simple to negation, existence, and wild cards—all without paying a noticeable performance penalty.

This paper discusses an implementation of LDAP that uses the IBM DATABASE 2* (DB2*) relational database as the data store and query engine to meet the directory service requirements and demonstrates performance via a set of benchmarks that categorize the types of queries. In detail, this paper examines the salient points of the University of Michigan reference implementation, the trade-offs of using a relational database, the LDAP hierarchical information model through a set of relations (tables), the mapping of the LDAP query language to SQL (Structured Query Language) given the defined relations, the access control model to provide authorization in the LDAP directory service, an identified set of items implemented to further enhance performance, and finally the performance comparison of a relational implementation to a file-system-based implementation given a set of well-defined benchmarks. This paper is an enhanced version of an earlier conference paper.[2]

Summary of related work

LDAP was originally implemented by the University of Michigan (U of M). The U of M reference implementation[3] is freely available through their FTP site. The implementation of the U of M LDAP is based on several freely available b-tree packages, such as GNU dbm and Berkeley db packages. This reference implementation supports LDAP version 2 protocol and is used as a basis for the LDAP/DB2 directory. The LDAP protocol and APIs are defined in IETF RFCs (Requests for Comments) 1777–1779 and 1823.[4–7]

LDAP was coauthored by Timothy A. Howes and Mark C. Smith,[8] and an implementation was done at the University of Michigan. The U of M LDAP is a sound reference implementation that helps people to understand the internals of LDAP. However, it needs a lot of enhancements to be a reliable and scalable enterprise directory service. First, the number of entries is limited. It does not scale to more than a few hundred thousand to possibly a million entries. By using simple file-system-based hash and b-tree packages, it is not able to handle large amounts of data. In contrast, relational database technologies such as DB2[9] are designed to handle terabytes of data. Second, populating directories with large numbers of entries is time-consuming work. A great amount of time is required to populate the directories with millions of entries. Third, because of limited search and indexing facilities provided by the file-system-based back end, only candidate entries can be retrieved. Then each entry is filtered through the filter program before it is returned to the client. However, in some cases, when the set of candidates is large, the search degenerates into a sequential search. For example, we discovered that negation queries and existence queries are fairly expensive with both the reference implementation and LDAP server from Netscape Communications Corporation. By using the powerful search engines provided by DB2, we were able to address some of the weak areas.

Mapping the LDAP model[1,10,11] to relational tables, however, is not a trivial task. First, LDAP allows both single-valued and multivalued attributes. But a relational database does not deal with multivalued attributes well. Second, the size of each DB2 table[9] is limited to 4K. Third, the LDAP model is hierarchical. It is known that hierarchies are very easy to represent in hierarchical databases (such as the Information Management System, or IMS*), where the structure of the data and the structure of the database are the same. Unfortunately, it is generally thought that relational databases do not provide adequate support for such data. It is not possible to directly map hierarchical data into tables because tables are based on sets rather than on graphs. Different vendors provide different mechanisms for the tree structure. For example, DB2[9] provides the WITH clauses in the SELECT statement to provide subtree traversal with arbitrary depth. Oracle[12] has CONNECT BY PRIOR and START WITH clauses in the SELECT statement to provide partial support for reachability and path enumeration. But all mechanisms will end up with recursive queries to handle hierarchical structures. Through experimentation, we discovered that recursive queries do not scale up well for large numbers of records in the table. We did a small experiment with 1000 LDAP entries using DB2 recursive queries; a simple select takes more than five minutes to complete.

U.S. Patent 5467471[13] presents a solution that does not require recursive query. The invention provides a genealogy table with which the directory hierarchy is represented in a table form. Each column of the genealogy table represents a level of the directory tree. This solution might be fine for directories with limited hierarchy depth. However, it is very difficult to realize the idea in practice when the directory is infinitely deep and the number of columns of a table is limited. We have attempted a similar implementation but learned that the complexity is very great and the performance implication is unclear.

To address this problem, we invented an efficient method[14,15] to represent LDAP hierarchies with relational tables without the overhead of recursive queries.[16–20] Our performance results showed that our implementation is competitive with other directory products in the industry.

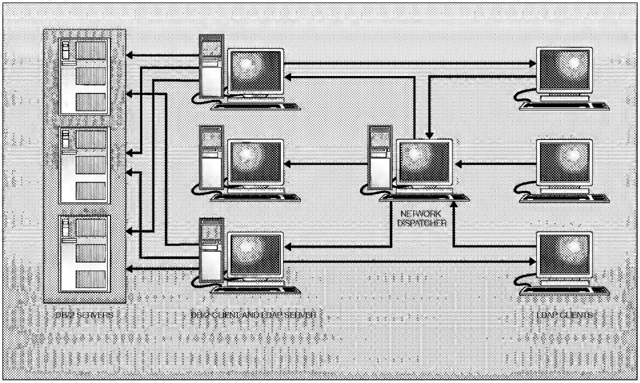
Trade-offs of using DB2

This section is a summary of the advantages of using DB2 as the LDAP data store.

One advantage is being a highly scalable data store. Unlike b-tree libraries in which the application process is tightly coupled with the physical storage, DB2 offers a number of solutions to store and retrieve large amounts of data. First of all, DB2 provides very flexible data placement options based on table spaces. A DB2 database can have multiple table spaces, and each table space can be created, modified, and recovered independently of another. Tables can be created in a preallocated storage (within a table space), or the storage can be allocated only when it is required or when the data are populated. The table data can be entirely contained in a single table space or multiple table spaces. DB2 also incorporates the concept of containers. A container is the allocation of physical spaces. In AIX* (Advanced Interactive Executive), a container is a logical volume. With the layer of table space, DB2 is able to handle large amounts of data independent of the limit of physical data storage (hard drive) and logical data storage (file system). For example, the maximum capacity of an AIX 4.1.x file system is 2 gigabytes. With table space, DB2 allows a database to grow up to 60 terabytes of data on a uniprocessor machine. Second, DB2 provides a truly distributed data store solution. Various configurations can be arranged with DB2. The configurations can be classified into the following categories:

- Single client/multiple server configuration. In a client/server configuration, every database resides on one network node, a database server, and is managed by a database management system (DBMS) running on that node. The applications can run on network nodes, such as the client nodes, separated from servers. The DB2 client provides the communication mechanism to retrieve data from the DB2 server. In the LDAP environment, the LDAP server (a database client) can connect to a number of networked databases that contain the directory information. However, from the user's perspective, the LDAP server actually stores all the information without knowing in which database the data are actually located. With this configuration, the LDAP server is freed from managing the physical data storage and is able to retrieve information from multiple database servers that work together to form a huge data store.
- Multiple clients/multiple servers configuration. The database clients can connect to any database server that contains the directory information. The collection of database servers actually form a single directory system image, and more than one

Figure 1 Multiple clients/multiple servers configuration



LDAP server can access the directory information. Since all the LDAP servers see the same directory image, a network dispatcher can be deployed to route requests among the LDAP servers. Figure 1 illustrates the configuration.

- Multiple clients/parallel super server configuration. In certain environments where users need to store large amounts of information in the directory, DB2 PE (Parallel Edition) provides a solution to enable users to store huge amounts of information in a single database. Instead of partitioning the database from the application level, DB2 PE automatically partitions the database into different machines. In addition, DB2 PE divides database queries into smaller, independent tasks that can execute concurrently; it is possible to complete the query fast enough to meet an end user's response time needs. LDAP can benefit from the power of parallel processing of DB2 PE with no changes to the server implementation. Another advantage of the parallel database solution is the ability to expand the system incrementally. This means users can have a startup system with smaller, lower-cost configurations to match initial size requirements. Then, as the database grows, users can easily add appropriately sized resources to accommodate growth. DB2 PE enables users to store terabytes of data into a database. Figure 1 illustrates the configuration.

A second advantage is being an atomic transaction. With the atomic transaction supported by DB2, LDAP servers can survive hardware or software failures and still maintain the integrity of the directory information. One of the problems with b-tree packages is that there are occasions where the database will become corrupted because hardware or software failures occur while updates are in progress. DB2 provides atomic transaction to ensure that updates of committed transactions will be recorded in the database. Uncommitted transactions, however, will be rolled back when system failure occurs. Database integrity will be maintained under all circumstances.

Figure 2 Possible syntaxes and meanings

```
bin: binary
dn: distinguished name
ces: case exact string
cis: case ignore string
tel: telephone number string (like cis but blanks and dashes are
    ignored during comparisons)
dn: distinguished name
```

As a third advantage, it is an on-line backup and restore facility. DB2 allows users to back up databases while it is either on line or off line. If the backup is performed off line, only the backup task can be connected to the database. If the backup is performed on line, other applications will be able to connect to the database while the backup task is running. The database can be local or remote. Users can back up a database or table space to disk, table, or a location managed by a utility such as the IBM ADSM (ADSTAR Distributed Storage Manager). A graphical tool is available (db2jobs) for users to monitor backup and recovery. DB2 provides a very fast and efficient restore facility. A 100K entry LDAP directory can be restored in less than 10 minutes.

As a fourth advantage, it provides alternative replication support. In addition to the replication supported by LDAP, the user can use the DB2 data replicator to provide faster and more efficient replication for frequent updates.

As a fifth advantage, it is a fast database loading facility. LDAP/DB2 provides a fast facility (bulkload) for initial load or for updating tables with large amounts of data. This facility is based on the DB2 LOAD utility. DB2 can move data into tables, create an index, and generate database statistics. The bulkload tool can populate the directory with 100K entries twice as fast as our nearest competitor.

The sixth advantage is having a powerful query processing engine that can deal with complicated queries. As is well-known, SQL is a very powerful query language. One of the problems that we observed in the U of M reference implementation is that it is not able to handle queries with the negation operator (!). The query usually degenerates into a linear search. LDAP/DB2, in contrast, is able to translate all LDAP queries into SQL and obtains timely and accurate results.

To summarize, DB2 provides hard-won advantages such as scalability, transaction integrity, backup and recovery, stability, and a powerful query processing engine. But all of the features mentioned above come with costs. Before our implementation, we had concerns about how the extra path length added by DB2 would affect the response time for LDAP queries. After extensive performance measurements, we discovered that LDAP/DB2 performs well compared to other LDAP products in the industry.

LDAP and relational model

In this section, we describe the information model and the relational aspects of the model for LDAP.

LDAP information model. The LDAP directory database consists of entries. Each entry is composed of one or more attributes. A type is associated with each attribute, and an attribute can have more than one value in an entry. The attribute type determines the syntax of the attribute. The syntax of an attribute determines how the data will be compared against the values in the query. In LDAP v2,[4–7,21] the possible syntaxes and their meanings are listed in Figure 2.

The LDAP schema is a collection of attribute definitions, object class definitions, and other information that a server uses to determine how and what to return for a given request. An attribute is information of a particular type and may have one or more associated values. A set of attributes is called an object class. A directory entry is an instantiation of one or more object classes, that is, a set of attributes. Attributes in a given object class are specified as mandatory or optional. Mandatory attributes are required to have values in the directory entry. Optional attributes, in contrast, do not have to exist for the directory entry of an object class. Figure 3 illustrates a possible definition of an object class called Person.

In object class Person, cn (common name), sn (surname), and objectclass are mandatory attributes; mail, phone, address, and fax are optional attributes.

LDAP entries are arranged in a tree structure that typically follows a geographical and organizational structure but is not limited to such structures. Each entry is uniquely identified by a distinguished name (DN). The formal definition of a distinguished name is given in RFC 1779.[6]

The functions provided by LDAP can be categorized as:

- Query: search and compare. These operations are used to retrieve information from the database. For the search function, the criteria of the search is specified in the search filter. The search filter is a Boolean expression that consists of attribute names, attribute values, and the Boolean operators AND represented by &, OR represented by |, and NOT represented by !. Users can use the filter to perform fairly complicated search operations. The filter syntax is defined in RFC 1960.[22]

  In addition to the search filter, users can also specify where the search starts in the directory tree structure. The starting point is called the base DN. The scope of the search can be a single entry (base-level search), the children of an entry (one-level search), or an entire subtree (subtree search). Although LDAP does not provide separate read and list operations, the search operation is used to provide these operations by setting the DN, scope, and filter appropriately.

- Update: add, delete, and modify. Users can use these functions to update the contents of the directory.

- Authentication: bind and unbind. LDAP supports a simple ID (identifier) and password authentication scheme. In the bind operation, the user can specify the ID and password, and the server will use this information to authenticate the client. If successful, the authentication is in effect for the life of that LDAP session, that is, until the corresponding unbind operation is issued.

- Rename: ModRDN. MoDRDN renames an existing directory entry.

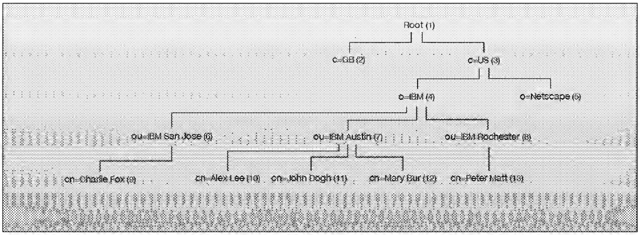
Figure 3 Possible definition of object class Person

```
objectClass Person
required cn,sn,objectClass
allows mail, phone, address, fax
```

Model LDAP through relations. At first glance, it seems very obvious that we should be mapping an LDAP object class into a DB2 relation. However, this mapping posed a serious problem since the LDAP model allows both single-valued and multivalued attributes. In the database design guideline, the First Normal Form requires that attributes within each tuple are ordered and complete and that the domains permit only simple values. Simple values cannot be decomposed into multiple values and cannot themselves be sets or relations. Some database systems (such as DB2) are attempting to support multivalued attributes. However, the implementation is not available yet. Unnormalized relations will make update operations (e.g., add, modify, and delete) fairly difficult to manage. We also discovered that we might lose some data semantics during the update process when multivalued attributes exist.

Instead, we mapped each LDAP attribute that can be searched by the user to an attribute relation. The attribute definitions were obtained from the attribute configuration files. This relation consists of two columns: unique entry identifier (EID) and normalized attribute value. In our system, each LDAP entry is assigned an EID. On the basis of the attribute syntax, the attributes are converted (or normalized) so that our system can apply SQL queries to the attribute values. For example, if the attribute syntax is case insensitive (CIS), the attribute value will be converted to all uppercase and stored in the attribute table. The attribute table is used mainly for the search operation to find the entries that match the filter crite-

Figure 4 LDAP naming hierarchy



ria. The actual entry data are stored in the ldap_entry table. In other words, the SQL queries generated by our system use the attribute table to locate the entry EIDs that match the filter expression and use the EIDs to retrieve the entry data from the ldap_entry table. Another advantage of this per-attribute table is that the size of the entry is no longer bounded by the DB2 4K limit. The attribute table and ldap_entry table are similar to the id2entry and attribute indices in the U of M reference implementation. The main difference is that our implementation is able to retrieve the exact target entries instead of just "candidates." As a result, no postprocessing of filtering entries is needed in LDAP/DB2.

A second challenge is to map LDAP to relational tables because the LDAP model is hierarchical. We discovered that with simple relations we were able to support LDAP search (base, one level, and subtree) with decent performance. The next subsection discusses the details.

In addition, we also have an LDAP entry table that holds the information about an LDAP entry. This table is used for obtaining the EID of the entry and supporting LDAP_SCOPE_ONELEVEL and LDAP_SCOPE_BASE search scope. Entries are stored using a simple text format of the form "attribute: value" as in the U of M reference implementation. Non-ASCII values or values that are too long to fit on a reasonably sized line are represented using a base 64 encoding. Given an ID, the corresponding entry can be returned with a single SELECT statement.

Mapping the LDAP hierarchy through relations. As illustrated in Figure 4, the LDAP naming hierarchy includes a number of entries where each entry is represented by a unique entry identifier (EID). Thus, for example, the root node has an EID = 1. Root has two children, entry GB ("Great Britain") with EID = 2, and entry US ("United States") with EID = 3. Child node US itself has two children, o = IBM (with EID = 4) and o = Netscape (with EID = 5). The remainder of the naming directory includes several additional entries at further sublevels.

A particular entry thus may be a "parent" of one or more child entries. An entry is considered a "parent" if it is located in the next higher level in the hierarchy. Likewise, a particular entry may be an ancestor of one or more descendant entries across many different levels of the hierarchy. A parent-child pair will also present an ancestor-descendant pair.

We created two relations to model the hierarchy in LDAP: parent-child (parent table) and ancestor-descendant (ancestor table). The parent table is created as follows. For each entry that is a parent of a child entry in the naming hierarchy, the unique identifier of the parent entry (PEID) is associated with the unique identifier of each entry that is a child of that parent entry. In the LDAP hierarchy illustrated in Figure 4, PEID 1 is associated with EID 2 and EID 3, PEID 3 is associated with EID 4 and EID 5, and so on. Each row of the parent table includes a PEID-EID pair.

Figure 5 One-level SQL query skeleton

```
One-Level Search:
    SELECT <data fields>
        from ldap_entry as entry
        where entry.EID in (
        select distinct ldap_entry.EID
        from ldap_entry as pchild, <table list>
        where ldap_entry.EID=pchild.EID AND pchild.PIED=<root dn id>
        AND ldap_entry.EID IN <sql select statements >)
```

The descendant table is created as follows. For each entry that is an ancestor of one or more descendant entries in the hierarchy, the unique identifier of the ancestor entry (AEID) is associated with the unique identifier of each entry that is the descendant (DEID) of that ancestor entry. The AEID field is the unique identifier of an ancestor LDAP entry in the LDAP naming hierarchy. The DEID field is the unique identifier of the descendant LDAP entry. Thus, in the naming hierarchy illustrated in Figure 4, AEID 1 has DEIDs 2–13 because each of the entries 2–13 is also a descendant of the root node. AEID 3 has DEIDs 4–13, AEID 4 has DEIDs 6–13, and so on. Each row in the descendant table thus includes an AEID-DEID pair.

For the LDAP search operation, the criteria of the search are specified in a search filter. The search filter is typically a Boolean expression that consists of attribute name, attribute value, and the Boolean operators AND, OR, and NOT. Users can use the filter to perform complex search operations. The filter syntax is defined in RFC 1960.[22] In addition to the search filter, users can also specify where in the directory tree structure the search is to start. The starting point is called the base DN. The search can be applied to a single entry (a base-level search), the children of an entry (a one-level search), or an entire subtree (a subtree search). Thus, the "scope" supported by an LDAP search consists of base, one level, and subtree. The parent and ancestor tables are used to facilitate one-level and subtree searches without recursive queries. In both cases, the search begins by going into the database and using the LDAP filter criteria to retrieve a list of entries matching the filter criteria. If the search is a one-level search, the parent table is then used to filter out EIDs that are outside the search scope (based on the starting point or base DN). Likewise, if the search is a subtree search, the descendant table is then used to filter out EIDs that are outside the search scope (again, based on the base DN). However, all the steps mentioned above are performed in a single SQL query. Figures 5 and 6 are some examples of the SQL query skeleton that we used during the one-level and subtree search. In these examples, <data fields> represents the SQL column name of the relations defined in the LDAP/DB2 schema that is described in more detail in the next subsection. <table list> and <sql select statements> are the two null terminated strings returned by the filter translator. The details of how the inner <sql select statements> were generated are described in the next section. <root dn id> is the unique identifier of the root DN.

In the one-level search query, the parent table information is contained in the ldap_entry table. Since the ldap_entry table also contains the entry information, we use SQL as operator to provide an alias pchild to represent the parent and child relation. Then, in the where clause, "ldap_entry.EID=pchild.EID" is used to filter out entries that are not in the one-level search scope.

In the subtree search query, the ancestor information is stored in the ldap_desc table. The inner where statement "ldap_entry.EID=ldap_desc.DEID" is used to filter out entries that are not in the subtree search scope.

Figure 6 Subtree SQL query skeleton

```
Subtree Search:

   SELECT <data fields>
      from ldap_entry as entry
      where entry.EID in (
      select distinct ldap_entry.EID
      from ldap_entry, ldap_desc, <table list>
      where (ldap_entry.EID=ldap_desc.DEID AND ldap_desc.AEID=<root dn id>)
      AND ldap_entry.EID IN <sql select statements >)
```

Database schema. We now give a detailed explanation of the tables that we defined in LDAP/DB2.

Entry table. The entry table holds the information about an LDAP entry. This table is used to obtain the EID of the entry and to support LDAP_SCOPE_ONELEVEL and LDAP_SCOPE_BASE search scope. The parent and child table is included in the entry table since all the other attributes are dependent on EID. The columns in this table are:

- EID—The unique identifier of the LDAP entry. This field is indexed.
- PEID—The unique identifier of the parent LDAP entry in the naming hierarchy. For example, the LDAP entry with the name "ou=Information Division, ou=People, o=University of Michigan, c=US" is the parent of "cn=Barbara Jensen, ou=Information Division, ou=People, o=University of Michigan, c=US."
- DN—The distinguished name of the entry.
- DN_TRUNC—Truncate DN to 250 characters so that we can build indices on this field.
- EntryData—Entries are stored using a simple text format of the form "attribute: value" as in the U of M reference implementation. Non-ASCII values or values that are too long to fit on a reasonably sized line are represented using a base 64 encoding. Given an ID, the corresponding entry can be returned with a single SELECT statement.
- Creator—The DN of the entry creator.
- Modifier—The DN of the entry modifier.
- modify_timestamp—Records the time when the entry was last modified.
- create_timestamp—Records the time when the entry was created.

Attribute table. There is one attribute table per searchable attribute. Each LDAP entry is assigned a unique identifier (EID) by the backing store. The columns for this table are:

- EID—The unique identifier of the LDAP entry.
- Attribute value—Normalized attribute values.
- Truncated attribute value—If the length of the column is longer than 250 bytes, a truncated column is created for indexing. In DB2, the maximum length for an indexed column is 255 bytes. The SQL type of the attribute depends on the LDAP data type. Indices can be created for attributes whose size is less than 255 bytes.

Descendant table. The purpose of the descendant table is to support the subtree search feature of LDAP. For each LDAP entry with a unique ID (AEID), this table contains the unique identifiers (DEID) of the descendant entries. The columns in this table are:

- AEID—The unique identifier of the ancestor LDAP entry. This entry is indexed.
- DEID—The unique identifier of the descendant LDAP entry. This entry is indexed.

For every entry in the directory, a row exists in this table for each of its ancestors including itself. The size of the table depends on the depth of each entry. In the worst case, if all the entries were at the same depth, the number of rows in the table is in $O(nm)$, where n is the number of nodes in the directory and m is the depth of the tree.

LDAP filter to SQL translation

This section discusses how LDAP/DB2 translates LDAP filters[22] into various types of SQL queries. We implemented a filter translator to generate the equivalent SQL expression corresponding to an LDAP filter that can be used in the WHERE clause of an SQL SELECT statement. For all queries, the general approach is to obtain the entry EIDs that match the search criteria based on the filter from the attribute table. Then the parent or ancestor tables are used to check whether the EIDs are located in the subtree under the base DN. After obtaining the entry EIDs that satisfy the filter and search scope criteria, the entry data are retrieved from the LDAP entry table. However, all the operations mentioned above are performed in a single SQL query. We discovered that combining subqueries into a single query is much more efficient than performing subqueries independently. The combined SQL query not only saves context switching cost, but also provides the DB2 query optimizer with more information to come up with an optimum access plan.

LDAP filters consist of six basic search filters with the format <attribute> <operator> <value>. Complex search filters can be generated by combining basic filters with the Boolean operators AND, OR, and NOT.

The skeleton of SQL SELECT statements used by LDAP/DB2 search routines are illustrated in Figure 7.

On the basis of the filter received, our SQL translator will generate <table list> (a list of attribute tables) and <sql select statements>. <root dn id> is the unique identifier of the root DN. The translation rules for basic filters and Boolean filters are presented in the following subsections. In the translation rules, tablename is the SQL table for the specified attribute, and columnname is the column name containing the attribute values.

Equality. The equality search operator locates entries with attributes exactly equal to the given value. The translation rule is shown in Figure 8.

For example, the purpose of the filter (sn = Jensen) is to find surnames exactly equal to Jensen. The corresponding SQL subquery generated for this filter is (SELECT EID FROM sn WHERE sn = 'jensen').

Ranges. For attributes supporting ordering, the LDAP filter provides inequality operators such as "greater than or equal" and "less than or equal." The translation rules are given in Figure 9.

For example, the LDAP filter (sn >= Jensen) locates entries with surnames lexicographically greater or equal to Jensen. The corresponding SQL subquery generated for this filter is (SELECT EID FROM sn WHERE sn = 'jensen').

Substring. LDAP supports arbitrary substring matching for text attributes. The user can put the wild-card character (*) at the beginning of a string, the middle of the string, the end of the string, or any combination of these in the LDAP filter. The format of the substring filter is:

```
(<attr> = [<leading>]* [any]*[<trailing>])
```

The SQL operator LIKE is used for substring matching. The SQL LIKE operator has the following syntax:

```
column LIKE PATTERN
```

PATTERN combines string constants with wild-card characters. SQL recognizes two wild-card characters: (1) the percent symbol (%), which means match zero or more characters, and (2) the underscore symbol (_), which means match any one character.

We use the SQL wild-card character "%" for the LDAP wild-card character. The LDAP substring filter "(attribute=value-with-stars)" is translated into "(SELECT EID FROM tablename WHERE columnname LIKE 'value with percents')."

For example, the LDAP filter (sn=*jensen*) locates surnames containing the string "jensen." The following SQL query is generated:

```
(SELECT EID FROM sn
  WHERE sn LIKE '%jensen%')
```

Approximate. The approximate search operator locates entries with attributes that sound like the given attribute value. The format of the approximate search filter is (<attr>~=<value>).

Figure 7 Skeleton of SQL SELECT statements

```
Base-Level Search:
   SELECT entry.EntryData, <operational attributes>
          from ldap_entry as entry
          where entry.EID in (
             select distinct ldap_entry.EID
             from <table list>
             where (ldap_entry.EID=<root dn id> )
             AND ldap_entry.EID IN <sql select statements >)

One-Level Search:
   SELECT entry.EntryData, <operational attributes>
          from ldap_entry as entry
          where entry.EID in (
             select distinct ldap_entry.EID
             from ldap_entry as pchild, <table list>
             where ldap_entry.EID=pchild.EID AND pchild.PIED=<root dn id>
             AND ldap_entry.EID IN <sql select statements > )

Subtree Search:
   SELECT entry.EntryData, <operational attributes>
          from ldap_entry as entry
          where entry.EID in (
             select distinct ldap_entry.EID
             from ldap_entry, ldap_desc, <table list>
             where (ldap_entry.EID=ldap_desc.DEID AND ldap_desc.AEID=<root dn id>)
             AND ldap_entry.EID IN <sql select statements > )
```

Figure 8 Translation rule for equality operator

```
LDAP filter:
   (<attr> = <value>)
SQL expression:
   (SELECT EID FROM tablename WHERE columnname = 'value')
```

The DB2 SOUNDEX library function is used for an approximate search. The SOUNDEX function returns a four-character string that is either a CHAR or VARCHAR. The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search for words with similar sounds.

Figure 9 Translation rules for inequality operators

```
LDAP filter:

   (<attr> >= <value>)
   (<attr> <= <value>)

SQL expression:

    (SELECT EID FROM tablename WHERE columnname >= 'value')
    (SELECT EID FROM tablename WHERE columnname <= 'value')
```

Figure 10 Translation of LDAP search filter

```
SELECT EID FROM tablename WHERE SOUNDEX(columnname)
  =SOUNDEX('value')))
```

Figure 11 Example of SQL query

```
SELECT EID FROM sn WHERE SOUNDEX(sn)
  =SOUNDEX('jensen'))))
```

The LDAP search filter "(attribute~=value)" is translated to the SQL query in Figure 10. For example, the LDAP filter (sn~=jensen) locates entries with surnames that sound like "jensen." The SQL query generated is illustrated in Figure 11.

The basic LDAP filter can be combined to form more complicated filters using the Boolean operators—AND (&), OR (|), and NOT (!)—and a prefix notation.

Others. The attribute values specified in the LDAP filters cannot contain unescaped left or right parenthesis characters. Escape combinations (backslash followed by any character) are translated as indicated in Figure 12.

Any single-quote characters found in the attribute value will be translated to two single-quote characters since the SQL value is enclosed in single-quote characters.

Complex queries. On the basis of the basic translation rules mentioned above, our biggest challenges are to provide an algorithm that can do the following:

Figure 12 Translation of escape combinations

```
\) will translate to ) in the SQL value
\( will translate to ( in the SQL value
\* will translate to * in the SQL value
\\ will translate to \ in the SQL value
\c will translate to \c in the SQL value where c is any other
characters other than ) or ( or * or \
```

Figure 13 Example of intuitive solution

```
LDAP filter:
     ldap filter (|(f1='v1')(f2='v2'))

SQL Query:
     SELECT EntryData
     FROM ldap_entry, f1,f2
     WHERE (f1.f1='f1value') OR (f2.f2='f2value')
     AND (ldapentry.EID=f1.UID)
     AND (ldapentry.EID=f2.UID)
     AND (ldapentry.EID IN
               SELECT DEID from ldapdesc
               WHERE PEID=<UID>)
```

- Combine the basic expressions to form a single SQL query that will retrieve the target entries that exactly match the search criteria
- Deal with complicated LDAP queries with infinite logical depth
- Deal with *all* logical operators efficiently

An intuitive solution is based on joining the attribute tables and applying the basic expressions to the attributes in the joined table. The LDAP AND and OR operators, in this case, can be translated into SQL AND and OR directly. In addition to the combined SQL expression, we need to include the JOIN condition based on EID. Figure 13 contains an example.

However, it is difficult to generalize this solution to handle the NOT operator. The LDAP NOT operator is basically used to locate entries that do not match the search criteria. A naive solution is to directly translate the LDAP NOT operator into an SQL NOT operator. A sample SQL query is illustrated in Figure 14.

The SQL query in Figure 14 does not yield the correct answer. Figure 15 illustrates the problem. There are five entries in this sample database (EID from 1 through 5) where the values of attribute f1 for entry 1, 2, 3, and 4 are v1. However, f1 of entry 4 is a multivalued attribute which has value v1 and foo. With the SQL statement in Figure 14, entry 4 is the answer. However, the correct answer should be entry 5. We discovered the following problems with the table join approach:

Figure 14 Sample SQL query for NOT operator

```
LDAP filter:
      ldap filter (!(f1='v1'))

SQL Query:
      SELECT EntryData
      FROM ldap_entry, f1,f2
      WHERE NOT (f1.f1='v1')
                     AND (ldap_entry.EID=f1.UID)
                     AND (ldap_entry.EID IN
                     SELECT DEID from ldapdesc
                     WHERE PEID=<UID>)
```

- If an entry does not contain the target attributes (for example, entry 5), this entry should not be selected by the SQL statement.
- Since an attribute can have multiple values in LDAP, the table join query will select the entry in which one of the values meets the criteria (for example, entry 4). But this entry should not be selected based on the LDAP filter.

One solution to these problems is to retrieve all the entries from the database and filter out the candidate entry as done by the Netscape server. However, for an LDAP directory with a large number of entries, it takes a very long time to obtain the results.

Another problem that we found with the intuitive solution mentioned above is that the OR operation does not perform well even for a small database with thousands of entries. Because it is using JOIN to combine the attribute tables and ldap_entry tables, DB2 will take a cross-product of all the rows in the attribute tables and the ldap_entry table for the OR operation. Even though most of the rows in the cross-product are irrelevant, the DB2 SQL engine dutifully reports all these rows, which are generally *much* more than are needed for the subquery evaluation.

Our solution is based on the concept of EID sets. First, generate an SQL subquery for each LDAP operator based on the basic translation rules. The SQL subquery will generate a set of entry EIDs that match the LDAP basic operation. If the LDAP logical operator is OR, use UNION to unite the sets generated from the subquery. If the LDAP operator is AND, use INTERCEPT to intercept the sets generated from the subquery. We experimented with two different ways to put together the SQL query based on the EID set concept.

Figure 15 Problem with SQL query

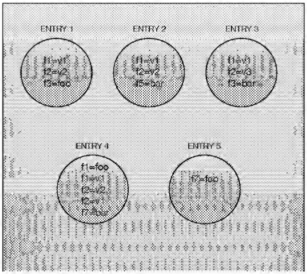


Figure 16 illustrates SQL queries that our system can generate for the LDAP filter (|(f1='v1')(f2='v2')).

Both SQL statements in Figure 16 generate the correct results. The first query is basically to perform the JOIN operation with the LDAP descendant table

Figure 16 SQL queries generated by system

```
Alternative 1:

  SELECT entry.EntryData
  FROM ldap_entry as entry
  WHERE entry.EID in
  (
    SELECT distinct ldap_entry.EID
    FROM ldap_entry, ldap_desc, f1
    WHERE
         (ldap_entry.EID=ldap_desc.DEID AND
          ldap_desc_AID=<id>) AND
          ldap_entry.eid=f1.eid AND
          f1='v1')
          UNION
          SELECT distinct ldap_entry.EID
          FROM ldap_entry, ldap_desc, f2
          WHERE (ldap_entry.EID=ldap+desc.DEID AND
                 ldap_desc.AEID=<id>)
                 AND ldap_entry.EID=f2.eid
                 AND f2='v2'))

Alternative 2:

  SELECT entry.EntryData
  FROM ldap_entry as entry WHERE entry.EID in
  ( SELECT distinct ldap_entry.EID FROM ldap_entry,ldap_desc
  WHERE
  (ldap_entry.EID=ldap_desc.DEID AND ldap_desc.AEID=<id>)
  AND  ldapentry.EID
  IN ((SELECT EID FROM f1 WHERE f1 = 'v1')
  UNION (SELECT EID FROM SN WHERE SN ='v2' )))
```

within each subquery. The second query is to perform the JOIN operation with the LDAP descendant table outside the subquery. Through extensive measurement, we chose to use Alternative 2 on the basis of the performance results. In addition to correct results, the OR operation performs reasonably well with both Alternatives 1 and 2 since relevant entries will be filtered out in the subquery, and target entries will be reported back to the main query.

With the set-based approach, the NOT operation can be performed by excluding entries with negation of the IN operation before the subquery. Figure 17 illustrates the operation.

With the basic translation rules and the EID sets approach, we implemented a recursive algorithm that can deal with complicated queries that have infinite logical operators. Figure 18 is an example of an SQL statement generated for a complex query with the AND, OR, and NOT operators.

Directory access control

Once information is stored in the directory, a method of protection must also be provided. When a user performs an operation on a directory entry, some mechanism must grant or deny permission to com-

Figure 17 NOT operation

```
Filter String:
  (!(f1='v1'))

SQL Statement:
  SELECT entry.EntryData,
  FROM ldap_entry as entry
  WHERE entry.EID in
  ( SELECT distinct LDAP_ENTRY.EID
    FROM ldap_entry,ldap_desc
    WHERE
    (ldap_entry.EID=ldap_desc.DEID AND ldap_desc.AEID=<id>)
    AND
    ldap_entry.EID NOT IN
    ((SELECT EID FROM f1 where f1='v1')))
```

Figure 18 SQL statement generated for complex query

```
Filter String:
  (&(|(objectclass=PERSON)(objectclass=GROUP))(sn=SMITH)(!(member=*)))

SQL Statement:
  SELECT entry.EntryData,
       FROM ldap_entry as entry WHERE entry.EID in
       ( SELECT distinct ldap_entry.EID FROM ldap_entry,ldap_desc
       WHERE
       (ldap_entry.EID=ldap_desc.DEID AND ldap_desc.AEID=?)
       AND
       ldap_entry.EID IN
       (((SELECT EID FROM OBJECTCLASS WHERE OBJECTCLASS = PERSON)
       UNION
       (SELECT EID FROM OBJECTCLASS WHERE OBJECTCLASS = GROUP))
       INTERSECT
       (SELECT EID FROM SN WHERE SN = SMITH)
       INTERSECT
       (SELECT EID FROM ldap_entry WHERE EID NOT IN
       (SELECT EID FROM MEMBER))))
```

plete the operation. This security concern is addressed by the access control model.

In designing the access control model, the primary goals were to create a simple, secure, and highly efficient means of protecting information stored within the LDAP directory. Using access control lists (ACLs), administrators can be assured that access to directory information is secure.

Administrators and users can view and update directory information through the LDAP protocol. The same is true for access control information (ACI). In order to use the standard LDAP operations to manage security, access control information is stored in attributes. By allowing updates to access control lists through the LDAP protocol, administrators are able to use a single protocol for all directory-related operations. The rest of this section provides the details of the access control mechanism that protects directory information.

Access control attributes. Each entry in the directory has an associated set of access control information. This information is stored in seven different attributes: aclEntry, entryOwner, aclPropagate, ownerPropagate, ownerSource, aclSource, and inheritOnCreate.

The attributes aclEntry and entryOwner describe the permissions given to particular subjects. The entryOwner attribute is multivalued, consisting of distinguished names (DNs) that are given full authority on an entry. These DNs are considered administrators for their corresponding entry. The aclEntry attribute is a multivalued attribute describing the user and group DNs that have been given privileges to perform particular operations on an entry. The information is associated with an entry in the LDAP directory. When an operation is initiated against that directory entry, the aclEntry and entryOwner are checked to determine whether the subject has the required permission to perform the requested operation on that particular directory entry.

Placing an ACL on each individual directory entry would be a very time-consuming exercise and would be difficult, if not impossible, to administer. Therefore, it is necessary to use a sparse ACL model. This means that not every node in the directory must have an ACL. Instead, ACLs are strategically placed in the tree and apply to entire portions of the tree instead of just a single node.

Although each directory entry has a set of ACL attributes that describe the security characteristics associated with it, it is not necessary that these attributes must be explicitly placed on the directory entry. ACL entries and entry owners set on a specific node can propagate to descendant entries. The access control policy can be managed on a per-directory entry basis, but it is also possible to manage entries in large portions of the tree with only a single ACL.

ACLs and an entry owner can be set to apply to just a particular entry, or an entry and its entire subtree. Although both entry owners and ACL entries can propagate, their propagation is not linked.

Entries on which an ACL has been placed are considered to have an "explicit" ACL. Similarly, if an entry owner has been set on a particular entry, that entry has an "explicit" owner. Since the two are not intertwined, an entry with an explicit owner may or may not have an explicit ACL, and an entry with an explicit ACL may or may not have an explicit owner. If these values are not explicitly present on an entry, the values are inherited from an ancestor node in the tree.

Each explicit ACL and entry owner applies to the entry on which it is set. Additionally, the values may apply to all descendants that do not have an explicitly set value. These values are considered "propagated"; their values propagate through the directory tree. Propagation of a particular value continues until another propagating value is reached. If an ACL (or entry owner) does not propagate, it is considered an "override" ACL (or entry owner) which applies only to the directory entry on which it is set.

The ACL of a directory entry can therefore be conceptually determined by the following algorithm: "Is there an explicit ACL set at the directory entry?" If yes, then that is the ACL of the directory entry. If no, then traverse the tree backwards until an ancestor node is reached with a propagate ACL. If no node is found with a propagate ACL, then only the entry owner and the administrator will be granted access to the directory entry.

The inheritOnCreate attribute ensures that directory entries are secure when they are created. At directory entry creation time, the creator may wish to specify attribute values for the access-control-related attributes. Whether or not the owner may specify ACL and ownership properties is determined by the parent directory entry's owner "inherit on create" flag.

Subjects. Subjects are comprised of an (LDAP) distinguished name and a privilege attribute type and represent a directory entry. The privilege attribute type describes whether the DN is an "access-id," a "group," or a "role." These different types enable access to be granted on both a per-user granularity as well as by group or role membership.

When the user authenticates to the directory at bind time, the subject information is retrieved. The DN used during authentication becomes the "bind DN." Additionally, group and role membership is determined and added to the list of credentials for the authenticated user. These credentials are later used by the access determination functions.

Two additional DNs have been created and are used as pseudosubjects. These pseudosubjects are not represented by an entry in the directory but can be used in an ACL. They are used to refer to large numbers of DNs that share a common characteristic at bind time in relation to either the operation being performed or the directory entry on which the operation is being performed.

The first pseudo-DN is the group "cn=Anybody." When specified as part of an ACL, this group refers to all users. Users cannot be removed from this group, and this group cannot be removed from the database. "cn=Anybody" is considered to be the group of all unauthenticated users or any user that does not have a specific ACL on a directory entry.

The second pseudo-DN is the access-id "cn=this." When specified as part of an ACL, it grants permissions when the bind DN matches the directory entry DN on which the operation is performed. If an operation is performed on the directory entry "cn=personA, ou=IBM, c=US," permissions associated with the access-id "cn=self" would be granted when the bind DN is "cn=personA, ou=IBM, c=US."

Permissions. The permission set is derived from the six basic LDAP operations that may be performed on a directory entry. The permissions are: add a directory entry, delete a directory entry, read an attribute value, write an attribute value, search for an attribute, and compare an attribute value. Each permission is discrete; one permission does not imply another.

Table 1 Permissions for LDAP operations

Operation	Permission Needed
ldapadd	add (on parent entry)
ldapdelete	delete (on entry)
ldapmodify	write (on attribute of entry)
ldapsearch: return attribute names	search (on attribute)
ldapsearch: return attributes and value	search, read (on attribute)
ldapmodrdn	write (on attribute)
ldapcompare	compare (on attribute)

Table 1 lists the permissions needed to perform each of the LDAP operations.

Add and delete permissions apply to an entire directory entry. Read, write, search, and compare apply to the attributes within the directory entry.

Attribute access classes. It is likely that many attributes will require the same type of protection. It is therefore useful to coarsen the access policy granularity by grouping attributes with similar access sensitivities. This action reduces the number of ACLs within the directory and greatly simplifies administration.

Attributes are grouped together in "attribute access classes." Within the schema file, attributes are mapped to an access control class. Each class is discrete; access to one attribute class does not imply access to another class.

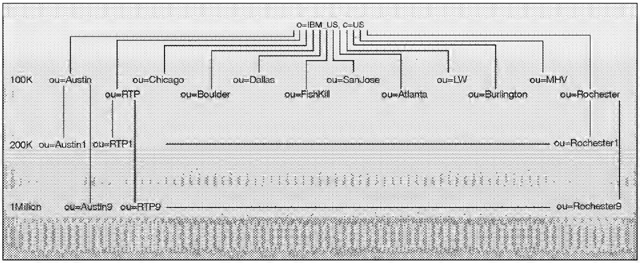
Instead of specifying that a subject has access to an attribute, the administrator gives a subject permission to an access class. This action grants the subject the specified permission to all attributes within that access class.

There are three access classes to which users can assign user modifiable attributes: normal, sensitive, and critical. Two additional attribute classes handle the special-case attributes. The restricted attribute class contains all of the ACL attributes, and the system attribute class contains all of the non-user-modifiable operational attributes.

Performance measurement

DB2 is a very powerful relational database product with a wide assortment of tools and tuning parameters that allow us to tailor our use of DB2 for optimum performance.

Figure 19 Base directory structure



Improvements. The following areas were key improvements in our LDAP/DB2 implementation with respect to performance:

- SQL query tuning. We found that although two similar queries could produce the same results in an entry-level directory (i.e., 10K entries), the tuned query performed much better when the directory was scaled to larger levels (i.e., 100K → 1M). The DB2 explain tool was used to tune our queries.
- Database indices. We also found that a single index on some of our tables did not scale well. We used a combined index for each LDAP attribute that was searchable, as well as a secondary single index for queries involving only the EID. The DB2 reorgchk was used to analyze the effectiveness of our indices and indicate when tables needed to be reorganized for better performance.
- Database partitioning. We found that splitting the database across multiple containers located on separate drives extended our scalability (avoiding operating system file size limitations) in addition to improving our search performance. The DB2 backup and redirected restore commands were used to experiment with different combinations of database partitioning and their effects on performance.
- Referral caching. We found that caching referral information at process startup time improved search performance because the information was already available when needed.
- Database and database manager tuning. We found the DB2 database monitor extremely useful for tuning our DB2 parameters for our specific directory contents.
- Server caching. We added a filter cache and an entry cache so that we could quickly determine whether a search had been done before, and if so, quickly return the results from memory instead of going to the database. We also added an ACL cache.
- Substring search improvements. We added a reverse index column in our attribute table to use when processing substring searches with a wildcard character "*" at the beginning.

Benchmark description. Because of a lack of a standard benchmark at the time, we created our own for internal development use and user-level competitive assessment. Our LDAP benchmark performs assorted ldapsearch queries for various entries contained in a directory. For this experiment, we used actual data from IBM's Callup (internal telephone) directory. A base of 100K (101498) unique entries was created by combining 12 IBM U.S. directories. Figure 19 illustrates our base directory structure.

Then, to scale up to 1 million entries, we made 10 copies of the data with an additional "ou" level added for uniqueness (Table 2). For example, if the first 100K contained Dave Bachmann in Austin, the second 100K would contain Dave Bachmann in Aus-

Figure 20 First time response time

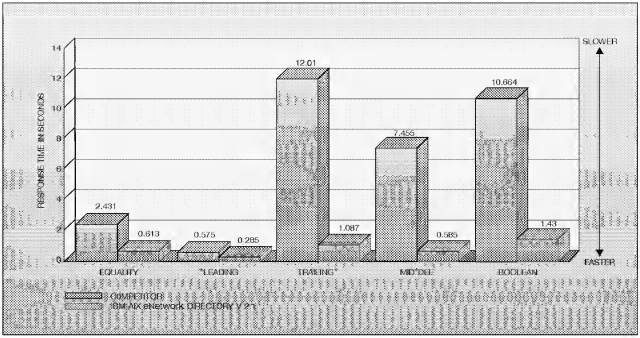


Table 2 Data copies for 1 million entries

Number of Entries	Respective DNs
100K	dn: cn=Dave Bachmann, ou=Austin, o=IBM_US, c=US
200K	dn: cn=Dave Bachmann, ou=Austin1, ou=Austin, o=IBM_US, c=US
300K	dn: cn=Dave Bachmann, ou=Austin2, ou=Austin, o=IBM_US, c=US
.	.
.	.
.	.
900K	dn: cn=Dave Bachmann, ou=Austin8, ou=Austin, o=IBM_US, c=US
1M	dn: cn=Dave Bachmann, ou=Austin9, ou=Austin, o=IBM_US, c=US

tin1, and the last 100K would contain Dave Bachmann in Austin9.

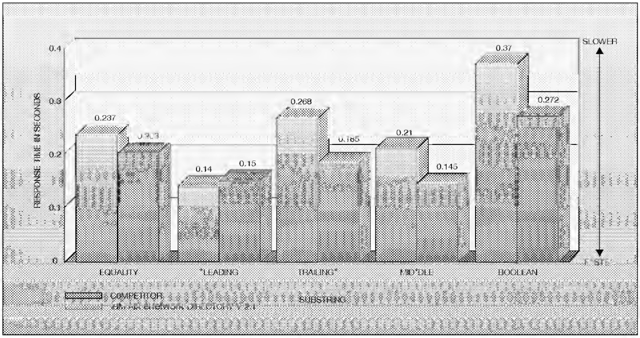
Each entry in the directory contains three required attributes and up to 43 additional attributes. On average, there are approximately 20 attributes per entry. We specified four indexed attributes in our LDAP configuration file (sn, cn, ou, telephone) and performed assorted ldapsearch operations with different filters and search scopes (base, one level, and subtree) on these attributes.

The hardware and software used for the experiment was as follows:

- Client—RS/6000* Model 6015 (601,66 MHz), 98 MB of memory, 1.5 GB DASD, 16 Mbps Token Ring, SPECint_base95=1.69, AIXv4.2.1, IBM eNetwork Directory Service v2.1 client
- Server—RS/6000 Model E20, 256 MB of memory, 1×2.2 GB and 4×4 GB disks, 16 Mbps Token Ring, SPECint_base95 = 3.43, AIX v4.3, IBM eNetwork Directory Service v2.1, DB2 v5

Benchmark results. Figures 20–23 illustrate our benchmark results. We have separated them into two categories: the *first* ldapsearch response times and the *cached* ldapsearch response times. First ldapsearch response times are important for appli-

Figure 21 Cache response time



cations and users performing random searches in the directory. A good example where first time searches are important would be a white pages directory. Cached ldapsearch response times are important for applications and users performing repeated searches in the directory. A good example where cached searches are important would be a public key infrastructure that stores certificates in the LDAP directory. For our benchmark results, we compared the AIX IBM eNetwork Directory v2.1 product to a competitive LDAP server (nonrelational) running on the same hardware.

For first time equality searches, the IBM average response time was four times better than the competitor we studied, as illustrated in Figure 20. Further analysis showing the distribution of response times with respect to the number of entries returned from the ldapsearch is shown in Figure 22. The competitor did not do well for any equality searches that returned more than one or two entries. For equality searches that returned more entries, the IBM response time increased only slightly and performed consistently well. We believe that this result shows the advantage of using a relational database for a directory data store. Although there is a certain amount of overhead associated with using a full-function database, its advantage becomes evident when returning large amounts of data.

For first time substring searches, response time varied depending on the location of the wild-card character "*." In a previous release, we found that we did not do well at all for this character at the beginning; we have since implemented a solution that performs much better. For searches with the character at the end or middle, we performed very well, beating our competitor significantly for every search in this category.

For Boolean first time queries (queries with AND and OR), the IBM average response time was seven times faster than that of the competitor. Further analysis of the searches showed that IBM did much better for searches that used the same attributes. For example, "(|(sn=Corn)(sn=Jones))" performed better than one that used two different attributes such as "(&(sn=Corn)(division=11))." This results from our attributes being stored in separate database tables. Similar to the equality results, IBM did much better when multiple results were returned. We did not include queries with the NOT operator in Figure 20 be-

Figure 22 LDAP first response time distribution

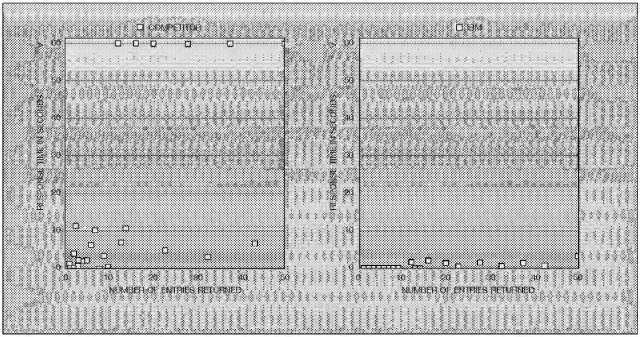
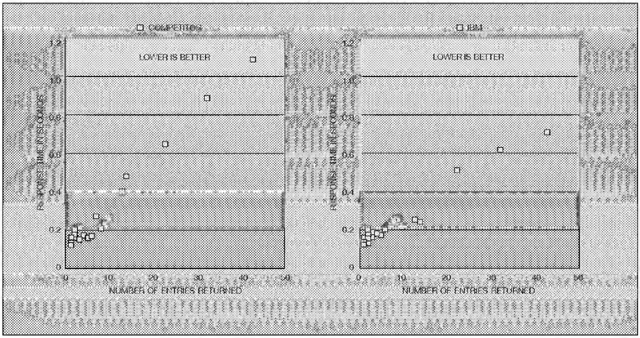


Figure 23 LDAP cache response time distribution



cause the competitive product takes a very long time to return any data. Our server returns correct data in seconds.

For cache searches, both products performed very quickly, regardless of the type of search being performed. This result is expected, since the data are being retrieved from memory instead of from the database. Further analysis showing the distribution of response times with respect to the number of cached entries returned from the ldapsearch is shown in Figure 23. For both products, the response time increased as the number of entries returned increased. Based on the data we have, the slope of the increase was twice as much for our competitor. For applications and users returning multiple entries, the IBM eNetwork Directory would be faster. We also experimented with committing and not committing SQL operations for search. We did not observe any performance differences.

Performance outlook. There are many more performance enhancements to our LDAP implementation that can further improve performance. Some examples are further caching improvements, table reduction, modification improvements, and lock reduction.

Conclusions

The design and implementation of an LDAP directory service using the IBM DB2 relational database as a data store and query engine has been examined. It has been shown that although the mapping of the LDAP hierarchical model to a set of relational tables (database schema) is a nontrivial task, a well-thought-out (yet simple) set of relational tables and use of SQL produces a secure directory service with competitive performance. Use of a relational database also provides:

- An easy growth path for a highly scalable directory server whose (1) database size on a single uniprocessor machine is not limited by the maximum file size, (2) client/server configuration allows for a network dispatcher to route requests among the LDAP servers for scaling to very large numbers of users while maintaining performance, and (3) parallel edition can automatically partition the database across multiple machines to support even more users and larger directories
- A robust data store whose transaction capability gives the LDAP server protection from hardware and software failures while maintaining the integrity of the directory information (no corrupted directory)
- Alternative administrative capabilities in areas that are not yet standardized by the LDAP such as access control, on-line backup and restore, replication, and a fast bulk load facility
- A powerful query processing engine to securely handle simple to complex queries using the well-known and powerful SQL language

Acknowledgments

Rod Mancisidor has inspired many of the ideas that we have presented in this paper. Larry Fichtner, Chin-Long Shu, Mark McConaughy, and Trung Tran also contributed greatly to the design and implementation. We would like to thank Dimitri Milionis and Bill Wilkins for providing tremendous help and advice with respect to DB2 performance and technical issues. We would also like to thank the management team (Reggie Hill, Sharal Brown, Dave Dyar, and Terry Dunkle) who provided us with strong support to deliver this product within a tight schedule.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Novell, Inc., Netscape Communications Corporation, Lotus Development Corporation, or QUALCOMM, Inc.

Cited references

1. T. Howes and M. Smith, *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*, ISBN 1-57870-000-0, Macmillan Technical Publishing, New York (1997).
2. S. Shi, E. Stokes, C. Corn, D. Bachmann, T. Jones, and S. Pasha, "Exploiting Relational Database Technologies with LDAP Directory Service," *Proceedings of the International Conference on Advanced Science and Technology* (1998), pp. 222–227.
3. *The SLAPD and SLURPD Administrator's Guide*, Release 3.3, University of Michigan, Ann Arbor, MI (April 30, 1996).
4. W. Yeong, T. Howes, and S. Kille, *Lightweight Directory Access Protocol*, RFC 1777, Internet Engineering Task Force (March 1995); available at http://www.ietf.org/rfc/.
5. T. Howes and M. Smith, *The LDAP Application Program Interface*, RFC 1823, Internet Engineering Task Force (August 1995); available at http://www.ietf.org/rfc/.
6. S. Kille, *A String Representation of Distinguished Names*, RFC 1779, Internet Engineering Task Force (March 1995); available at http://www.ietf.org/rfc/.
7. T. Howes, S. Kille, W. Yeong, and C. Robbins, *The String Representation of Standard Attribute Syntaxes*, RFC 1778, Internet Engineering Task Force (March 1995); available at http://www.ietf.org/rfc/.
8. T. A. Howes and M. C. Smith, *A Scalable, Deployable Directory Service Framework for the Internet*, CITI Technical Report 95-7, University of Michigan, Ann Arbor, MI (April 1995).

9. *DATABASE 2, Application Programming Guide for Common Servers*, S20H-4643-01, IBM Corporation.
10. J. J. Ordille and B. P. Miller, "Nomenclature Descriptive Query Optimization for Large X.500 Environments," *Proceedings of the 1991 SIGCOMM Conference* (1991), pp. 301–314.
11. D. Barrowman and P. Martin, "The Performance of SQL Queries in X.500 Directory System," *Computer Communications* **21**, 133–146 (1998).
12. *Oracle 7 Server SQL Reference Manual*, Oracle Corporation, see http://technet.oracle.com/docs/products/oracle7/doc_index.htm.
13. *Maintaining Databases by Means of Hierarchical Genealogical Table*, United States Patent, No. 5467471 (November 14, 1995).
14. S. S. B. Shi, L. G. Fichtner, R. A. Mancisidor, and C. Corn, *An Efficient Implementation of Lightweight Directory Access Protocol (LDAP) Search Queries with SQL*, filed as Docket AT998183 (1998).
15. S. S. B. Shi, L. G. Fichtner, R. A. Mancisidor, and C. Corn, *Method of Hierarchical LDAP Searching with Relational Tables*, filed as Docket AT998106 (1998).
16. J. Teuhola, "An Efficient Relational Implementation of Recursive Relationships Using Path Signatures," *The 10th International Conference on Data Engineering*, Houston, Texas (February 1994), pp. 446–454.
17. R. Agrawal and H. V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations," *Proceedings of the 13th VLDB Conference*, Brighton, England (1987), pp. 255–266.
18. "An Amateur's Introduction to Recursive Query Processing Strategies," *Proceedings of ACM SIGMOD Conference*, Washington, DC (1986), pp. 16–52.
19. P. Ciaccia, D. Maio, and P. Tiberjo, "A Method for Hierarchy Processing in Relational Databases," *Information Systems* **14**, No. 3, 93–105 (1989).
20. E. H. Herrin and R. A. Finkel, "Schema and Tuple Trees: An Intuitive Structure for Representing Relational Data," *Computing Systems: The Journal of the USENIX Association* **9**, No. 2, 93–118 (1996).
21. T. A. Howes, *The Lightweight Directory Access Protocol: X.500 Lite*, CITI Technical Report 95-8, University of Michigan, Ann Arbor, MI (July 27, 1995).
22. T. Howes, *A String Representation of LDAP Filters*, RFC 1960, Internet Engineering Task Force (June 1996); available at http://www.ietf.org/rfc/.

Accepted for publication September 15, 1999.

Shepherd S. B. Shi *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: sshi@us.ibm.com).* Dr. Shi is a senior software engineer and chief developer for IBM eNetwork Directory Services. He has a B.S. in computer science from National Taiwan University, an M.S. in computer science from Stanford University, and a Ph.D. in computer science from the University of Illinois.

Ellen Stokes *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: ejstokes@us.ibm.com).* Ms. Stokes is a senior technical staff member and the lead architect for Directory Services across IBM. She has a B.S.E. in computer engineering and an M.S.E. in computer, information, and control engineering from the University of Michigan.

Debora Byrne *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: djbyrne@us.ibm.com).* Ms. Byrne is a staff software engineer and is team lead for the SecureWay Directory User Interface. She has a B.S. in computer science from Duke University and an M.S. in software engineering from the University of Houston—Clear Lake.

Cindy Fleming Corn *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: ccorn@us.ibm.com).* Ms. Corn is a staff software engineer and the IBM Directory Performance lead member. She has a B.S. in computer science and mathematics from the University of Illinois.

David Bachmann *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: dbachman@us.ibm.com).* Dr. Bachmann is a senior software engineer and the team lead for Distributed Systems Performance. He has a B.S. in computer science and mathematics from Iowa State University, and an M.S. and Ph.D. in computer science and engineering from the University of Michigan.

Tom Jones *Innosoft International, Inc., 8911 Capital of Texas Highway, Suite 4140, Austin, Texas 78759 (electronic mail: tom.jones@innosoft.com).* Mr. Jones was a software engineer and member of the SecureWay Directory Performance team at the IBM Network Computing Software Division in Austin before joining Innosoft. He has eighteen years of computer science experience, the last five of which have been in performance analysis for distributed systems. Mr Jones is currently a senior quality assurance engineer at Innosoft International. He is responsible for product testing of the Innosoft Distributed Directory Server, the Innosoft DirectoryPortal, and the Innosoft LDAP Proxy Server, with much of his work including LDAP performance testing.